FactGuardian 长文本"事实卫士"智能体实验报告

小组成员：马舒童 10235501462； 张欣扬 10235501413； 詹江叶煜 10235501471（具体分工见分工文档分工.md）

一、项目概述

1.1 研究背景与痛点

在当今学术研究和商业报告编写领域，长文档（如毕业论文、可行性报告、项目方案）常常面临以下严峻挑战：

痛点场景	具体问题	影响后果
多人协同写作	不同章节由不同人员撰写，容易出现数据引用不一致	论文逻辑混乱，结论可信度下降
分章节生成	前后文对同一指标的描述产生冲突	报告自相矛盾，专业性受质疑
版本迭代	修改过程中数据更新不及时	前后数据对不上，引发质疑
引用错误	对外部数据的理解产生偏差	事实性错误，损害学术声誉

1.2 解决方案

FactGuardian 是一个云原生智能代理系统，专为长文本事实一致性验证而设计。系统作为"中间件"部署，自动完成：

1. **文档解析** → 支持多种格式（DOCX、PDF、TXT、Markdown）
2. **事实提取** → 基于 LLM 提取关键事实、数据点和结论
3. **冲突检测** → 自动检测内部逻辑冲突和不一致
4. **溯源校验** → 通过外部搜索验证事实真实性
5. **可视化分析** → 提供直观的 Dashboard 仪表盘

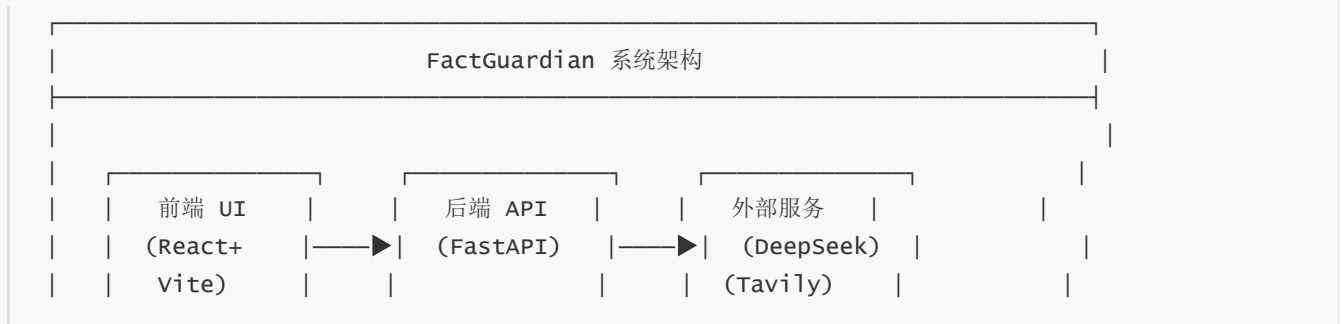
   此外，我们还根据实际使用需要额外补充了两个功能（具体见3.7与3.8）：

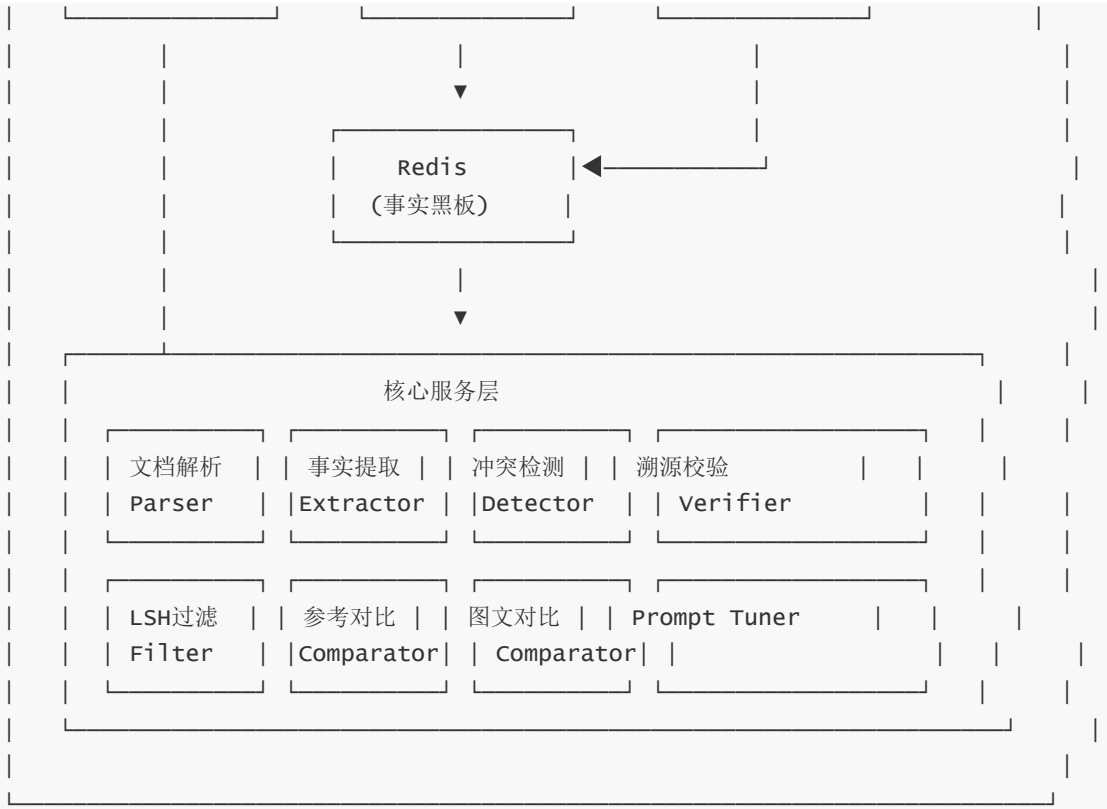
6.**参考文本对比功能**：上传参考文档，检测主文档与参考内容的相似度/引用关系

7.**图片/框架图对比**：上传框架图，检测文档描述与图片的一致性

二、技术架构设计（30%）

2.1 整体架构图





2.2 云原生组件运用

2.2.1 Redis 作为"事实黑板"

系统采用 Redis 作为统一的事实存储中间件，实现以下核心功能：

Redis 数据模型设计：

Key 模式	数据类型	用途	TTL
`facts:{document_id}`	Hash	存储提取的事实列表	24小时
`doc:{document_id}`	Hash	存储文档元数据	24小时
`conflicts:{document_id}`	Hash	存储检测到的冲突	24小时
`verifications:{document_id}`	Hash	存储校验结果	24小时

内存后备机制（Memory Fallback）：

```
# 模块级全局变量：共享内存后备存储（确保所有 RedisClient 实例使用同一个字典）
_SHARED_MEM_FACTS = {}
_SHARED_MEM_DOCS = {}
_SHARED_MEM_CONFLICTS = {}

class RedisClient:
    """Redis 客户端封装（单例模式）"""

    def __init__(self):
        # ...
```

```
        # 内存后备存储引用全局共享变量
        self._mem_facts = _SHARED_MEM_FACTS
        self._mem_docs = _SHARED_MEM_DOCS
        self._mem_conflicts = _SHARED_MEM_CONFLICTS
        # ...
```

设计优势：

- ✅ 单例模式确保全局只有一个 RedisClient 实例
- ✅ 内存后备机制保证服务可用性
- ✅ TTL 自动过期，节省资源
- ✅ 支持并发访问，适合分布式部署

2.2.2 Docker 容器化部署

后端 Dockerfile：

```
FROM python:3.10-slim

WORKDIR /app

ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1

RUN apt-get update && apt-get install -y --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Docker Compose 编排：

```
version: '3.8'

services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
      - REDIS_DB=0
      - DEEPSEEK_API_KEY=${DEEPSEEK_API_KEY}
```

```
    depends_on:
      - redis
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    restart: unless-stopped

volumes:
  redis_data:
```

2.3 技术选型合理性分析

组件	选型	理由
后端框架	FastAPI	高性能、异步支持、自动生成 API 文档
LLM	DeepSeek Chat	性价比高、中文效果好、API 稳定
搜索	Tavily/Serper	专业的事实核查搜索 API
缓存	Redis	高性能、持久化、支持数据结构
前端	React + Vite	组件化开发、热更新快、构建效率高
样式	Tailwind CSS	原子化 CSS、开发效率高、响应式支持
NLP	jieba + datasketch	中文分词、LSH 相似度计算

2.4 稳定性设计

1. 错误处理与异常捕获

- 多层 try-except 保护：API 端点、服务层、外部调用均有异常捕获

```
batch_results = await asyncio.gather(
    *[task for _, _, task in tasks],
    return_exceptions=True  # 单个失败不影响其他
)
# 处理结果
for (idx, fact, _), result in zip(tasks, batch_results):
    if isinstance(result, Exception):
        logger.error(f"Verification failed for fact {idx}: {str(result)}")
        # 添加错误结果而不是崩溃
        results.append({
            "fact_index": idx,
            "is_supported": None,
            "confidence_level": "Low",
            "assessment": f"验证过程出错: {str(result)}",
```

```
            ...
        })
```

- JSON 解析容错：多策略提取并处理格式错误

```
# 策略1: 寻找 markdown 代码块
# 策略2: 寻找最外层的 {}
try:
    parsed_result = json.loads(content_to_parse)
except (json.JSONDecodeError, ValueError) as e:
    logger.error(f"Failed to parse verification result: {e}")
    # 返回默认结果而不是崩溃
    parsed_result = {
        "is_supported": None,
        "confidence_level": "Low",
        "assessment": "模型输出格式错误，无法解析。",
        ...
    }
```

2. 降级策略（Fallback）

- Redis 降级到内存：Redis 不可用时使用内存后备

```
try:
    self.client.set(key, value)
    logger.info(f"保存事实成功...")
    return True
except Exception as e:
    logger.error(f"保存事实失败: {str(e)}，改用内存后备存储")
    self._mem_facts[document_id] = facts  # 内存后备
    return True
```

- LLM 不可用时的占位返回

```
if not self.llm_client.is_available():
    logger.warning("LLM not available, returning mock verification result")
    return {
        "is_supported": False,
        "confidence_level": "Low",
        "assessment": "LLM服务不可用（未配置 API Key），无法进行智能校验。仅作为占位返回。",
        ...
    }
```

- 搜索服务多提供商：Tavily → Serper → Mock LLM

```
self.provider = "mock"
if self.tavily_key:
    self.provider = "tavily"
elif self.serper_key:
    self.provider = "serper"
# 如果都不可用，使用 LLM Mock 搜索
```

3. 服务可用性检查

- 启动时检查 Redis 连接

```
try:
    check_client = redis.Redis(host=self.host, port=self.port, db=self.db,
socket_timeout=1)
    check_client.ping()
    logger.info(f"Redis 连接检查通过...")
except Exception as e:
    logger.warning(f"Redis 连接初始化检查失败: {e}")
    # 提供详细的环境配置警告
```

- API 端点前置检查

```
if not llm_client.is_available():
    raise HTTPException(
        status_code=503,
        detail="LLM 服务不可用，请检查 DEEPSEEK_API_KEY 是否已配置"
    )
```

- 健康检查端点

```
@app.get("/health")
async def health_check():
    redis_status = "connected" if redis_client.is_connected() else "disconnected"
    llm_status = "configured" if llm_client.is_available() else "not_configured"
    return {
        "status": "healthy",
        "redis": redis_status,
        "llm": llm_status
    }
```

4. 超时与资源限制

- HTTP 请求超时

```
async with httpx.AsyncClient(timeout=60.0) as client:
    response = await client.post(url, json=payload, headers=headers)
```

- SSE 心跳保持连接

```
try:
    data = await asyncio.wait_for(queue.get(), timeout=10.0)
    yield f"data: {json.dumps(data, ensure_ascii=False)}\n\n"
except asyncio.TimeoutError:
    # 发送心跳保持连接
    yield f": heartbeat\n\n"
```

- 批量处理限制

```
MAX_AUTO_VERIFY = 200
for i, fact in enumerate(all_facts):
    if i >= MAX_AUTO_VERIFY:
        logger.warning(f"Reached verification limit {MAX_AUTO_VERIFY}, skipping rest")
        break
```

5. 日志记录

- 分级日志（info/warning/error/debug）
- 关键操作记录（上传、提取、验证、冲突检测）
- 错误详情记录便于排查

2.5 扩展性设计

1. 模块化架构

- 服务分离：Parser、FactExtractor、ConflictDetector、Verifier、SearchClient 等独立模块
- 单一职责：每个服务专注单一功能
- 依赖注入：通过构造函数注入依赖，便于测试和替换

2. 单例模式

- 全局服务实例：避免重复创建，统一管理

```
# 全局 Redis 客户端实例
redis_client = RedisClient()
```

- 共享内存后备：模块级全局变量确保一致性

```
# 模块级全局变量：共享内存后备存储
_SHARED_MEM_FACTS = {}
_SHARED_MEM_DOCS = {}
_SHARED_MEM_CONFLICTS = {}
```

3. 配置管理

- 环境变量配置：API Key、服务地址等通过环境变量管理

```
self.api_key = os.getenv("DEEPSEEK_API_KEY")
self.base_url = os.getenv("DEEPSEEK_BASE_URL", "https://api.deepseek.com")
```

- 默认值支持：提供合理的默认配置

4. 异步并发处理

- 批量并行验证

```
batch_size = 10  # 每批并行处理10个事实
batch_results = await asyncio.gather(
    *[task for _, _, task in tasks],
    return_exceptions=True
)
```

- FastAPI 异步端点：支持高并发请求

5. 缓存机制

- Redis 缓存：事实、文档元数据、冲突结果
- TTL 管理：自动过期（24小时）

```
# 设置过期时间（24小时）
self.client.expire(key, 86400)
```

6. 进度管理

- SSE 实时推送：支持长时间任务的进度跟踪
- 进度管理器：统一管理多文档处理进度

```
@app.get("/api/progress/{document_id}")
async def stream_progress(document_id: str):
    async def event_generator():
        queue = progress_manager.subscribe(document_id)
        # SSE 推送进度更新
```

7. 可插拔设计

- 多搜索提供商支持：Tavily、Serper、Mock
- 多 Vision API 支持：OpenAI、Claude、豆包

```
if self.doubao_key:
    logger.info("使用豆包 Vision API")
elif self.anthropic_key:
    logger.info("使用 Claude Vision API")
elif self.openai_key:
    logger.info("使用 OpenAI Vision API")
```

8. 数据结构扩展性

- 灵活的事实结构：支持动态字段（subject、predicate、object、value、modifiers 等）
- JSON 存储：便于扩展新字段

三、智能逻辑实现（30%）

3.1 事实提取模块

3.1.1 结构化事实 Schema

系统设计了完善的事实数据模型，确保提取结果的一致性和可扩展性：

```
DEFAULT_FACT_KEYS = {
    "subject": None,           # 主体（谁/哪个项目）
    "predicate": None,         # 谓词（关系/动作）
    "object": None,            # 客体（目标）
    "value": None,             # 数值
    "modifiers": {},           # 修饰符（单位、范围等）
    "time": None,              # 时间
    "polarity": "affirmative", # 极性（肯定/否定）
    "type": "未知",            # 类型（数据/日期/人名/结论/事件）
    "verifiable_type": "public",  # 可验证类型（public/internal）
    "confidence": 0.0,         # 置信度
    "location": {},            # 位置信息
}
```

3.1.2 LLM Prompt 工程

材料驱动提示优化（Prompt Tuner）：

```
UNITS_PATTERNS = [
    r"\%", r"万元|人民币|元|美元|万元人民币|亿|万",
    r"人|户|家|台|件|公里|米|平方米|亩",
]
TIME_PATTERNS = [
    r"\d{4}年\d{1,2}月\d{1,2}日", r"\d{4}年\d{1,2}月",
    r"\d{4}年", r"\d{1,2}月\d{1,2}日",
    r"\d{4}-\d{1,2}-\d{1,2}", r"\d{4}-\d{1,2}",
    r"\d{4}/\d{1,2}/\d{1,2}",
    r"月底|年初|年末|上半年|下半年|季度|Q\d",
]

class PromptTuner:
    def derive_hints_from_text(self, text: str) -> Dict[str, Any]:
        """从文本中提取关键词、单位、时间短语等提示信息"""
        # 提取领域关键词
        tokens = re.findall(r"[\u4e00-\u9fa5]{2,}|[A-Za-z]{2,}", text)
        keywords = list({t for t in tokens if len(t) >= 2})[:20]
        # 提取单位
        units = list({u for u in re.findall(pat, text) for pat in UNITS_PATTERNS})
        # 提取时间短语
        times = list({t for t in re.findall(pat, text) for pat in TIME_PATTERNS})
        return {"keywords": keywords, "units": units, "time_phrases": times}
```

事实提取 Prompt：

```
SYSTEM_PROMPT = """你是一个专业的事实提取助手。你的任务是从给定文本中准确提取关键事实信息，
并以结构化字段输出，便于后续一致性/冲突检测。

提取原则：
1. 提取完整事实，避免碎片化
2. 去除重复
3. 识别可验证性（public vs internal）
4. 结构化字段：subject/predicate/object/value/modifiers/time/polarity

verifiable_type 判定规则：
- "public"：已发生事件、已公开数据、已发布政策、可观测客观事实
- "internal"：未来计划、主观评价、内部数据、规划措施
"""
```

并行化提取优化：

```
async def extract_from_document(self, document_id, sections, ...):
    all_facts = []
    batch_size = 5  # 每批并行处理5个章节

    for batch_start in range(0, len(sections), batch_size):
        batch = sections[batch_start:batch_start + batch_size]

        # 并行提取事实
        tasks = []
        for idx_in_batch, section in enumerate(batch):
            idx = batch_start + idx_in_batch
            tasks.append(self.llm.extract_facts(...))

        results = await asyncio.gather(*tasks, return_exceptions=True)
        # 处理结果...

    return result
```

3.2 冲突检测模块

3.2.1 多策略冲突检测

系统实现了三种互补的冲突检测策略：

策略一：结构化字段驱动比对

```
def _generate_structured_pairs(self, facts, limit=30):
    """
    基于结构化字段生成候选对：
    - 同一 (subject, predicate, object) 分组内：
      * 极性相反 → 逻辑矛盾候选
      * 数值冲突 → 数据不一致候选
      * 时间冲突 → 时间不一致候选
    """
    def key_of(f):
        return (
```

```
            (f.get("subject") or "").strip(),
            (f.get("predicate") or "").strip(),
            (f.get("object") or "").strip(),
        )

    # 数值冲突检测（阈值：比例差≥10%，一般数值相对差≥20%）
    if va is not None and vb is not None:
        if has_percent_a or has_percent_b:
            if abs(va - vb) >= 10.0:
                pairs.append((fa, fb))
        else:
            if (min(va, vb) > 0 and abs(va - vb) / max(va, vb) > 0.2) or abs(va - vb) >
1.0:
                pairs.append((fa, fb))

    # 时间冲突检测
    if ta and tb and ta != tb:
        pairs.append((fa, fb))
```

策略二：关键词模式匹配

针对典型矛盾场景预设检测规则：

```
def _generate_keyword_based_pairs(self, facts, limit=30):
    """
    针对用户列出的典型矛盾进行模式匹配：
    - 合规/不合规（落实政策 vs 不符合新版指南）
    - 居民协调完成 vs 居民反对/延迟
    - 资金缺口（无缺口 vs 停工风险）
    - 竣工时间（可能延迟至4月 vs 调整为3月20日）
    - 医疗预约闭环 vs 无法对接/仍需线下
    - 前期筹备全部完成 vs 未办理施工许可证
    - 装修进度/费用比例不匹配
    - 安全目标 vs 技术问题
    """
    candidates += pairs_for([
        "落实国家及省级政策", "符合政策", "落实政策"
    ], [
        "不符", "未达到指南要求", "修订版"
    ])
    # ... 更多模式匹配
```

策略三：MinHash LSH 相似度过滤（性能优化）

```
class LSHFilter:
    """
    使用 MinHash + LSH 快速找到相似的事实对
    时间复杂度从 O(n²) 优化到接近 O(n)
    """

    def __init__(self, num_perm=128, threshold=0.3, num_shingles=2):
        self.num_perm = num_perm
```

```
        self.threshold = threshold
        self.num_shingles = num_shingles

    def filter_similar_pairs(self, facts, max_pairs=50):
        if DATASKETCH_AVAILABLE:
            return self._filter_with_minhash_lsh(facts, max_pairs)
        else:
            return self._filter_with_simple_similarity(facts, max_pairs)

    def _filter_with_minhash_lsh(self, facts, max_pairs):
        """使用 MinHash LSH 过滤"""
        lsh = MinHashLSH(threshold=self.threshold, num_perm=self.num_perm)
        minhashes = {}

        for i, fact in enumerate(facts):
            text = self._get_fact_text(fact)
            tokens = self._tokenize(text)  # 使用 jieba 分词
            shingles = self._get_shingles(tokens, self.num_shingles)

            m = MinHash(num_perm=self.num_perm)
            for shingle in shingles:
                m.update(shingle.encode('utf-8'))

            minhashes[f"fact_{i}"] = (m, fact, i)
            lsh.insert(f"fact_{i}", m)

        # 查找相似对
        pairs = []
        for fact_id, (m, fact, idx) in minhashes.items():
            similar_ids = lsh.query(m)
            for sim_id in similar_ids:
                if sim_id != fact_id:
                    pairs.append((fact, minhashes[sim_id][1]))

        return pairs[:max_pairs]
```

LSH 性能提升数据：

指标	原始 O(n²)	LSH 优化后	提升倍数
100 条事实	4950 对	~50-100 对	50-100x
500 条事实	124,750 对	~200-300 对	400-600x
处理时间	5-10 分钟	15-30 秒	10-20x

3.2.2 冲突分类与严重程度

```
CONFLICT_DETECTION_PROMPT = """以下是从同一文档不同位置提取的两个事实，请判断它们是否存在冲突。

请仔细分析，判断是否存在冲突（数据不一致、逻辑矛盾、时间冲突等）。

返回 JSON 格式：
{"has_conflict": true/false, "conflict_type": "无冲突/数据不一致/逻辑矛盾/时间冲突",
 "severity": "无/低/中/高", "explanation": "简短说明", "confidence": 0.5}"""
```

3.3 溯源校验模块

3.3.1 Chain of Thought 推理

```
VERIFICATION_PROMPT_TEMPLATE = """
请验证以下事实的真实性，采用思维链（Chain of Thought）方式分析：

1. 分析事实的核心主张（主体、谓词、客体、时间等）
2. 将核心主张与搜索到的信息进行比对
3. 检查是否存在矛盾或确认的证据
4. 综合判断置信度

最后输出 JSON 格式结果：
```json
{{
  "is_supported": true/false,
  "confidence_level": "High/Medium/Low",
  "assessment": "分析结论",
  "correction": "修正建议"
}}
```

"""
```

3.3.2 多源搜索集成

```python
class SearchClient:
    """支持 Tavily、Serper 或 Mock 模式的搜索客户端"""

    def __init__(self):
        self.tavily_key = os.getenv("TAVILY_API_KEY")
        self.serper_key = os.getenv("SERPER_API_KEY")
        self.provider = "tavily" if self.tavily_key else "serper" if self.serper_key else
"mock"

    async def search(self, query, max_results=3):
        if self.provider == "tavily":
            return await self._search_tavily(query, max_results)
        elif self.provider == "serper":
            return await self._search_serper(query, max_results)
```

```
        else:
            # Mock 模式：使用 LLM 模拟搜索结果
            return await self._search_mock_with_llm(query)

    async def _search_mock_with_llm(self, query):
        """使用 LLM 生成模拟搜索结果，降低测试门槛"""
        prompt = f"""请模拟搜索引擎的功能。针对查询 "{query}"，
        请生成 3 个看起来真实的搜索结果摘要。

        要求：
        1. 内容必须是准确、客观的事实
        2. 如果查询包含明显的事实错误，搜索结果应包含正确信息
        """
        # 返回模拟结果...
```

3.4 异常处理与鲁棒性

```python
async def chat(self, messages, model="deepseek-chat", temperature=0.3, max_tokens=4096):
    try:
        async with httpx.AsyncClient(timeout=60.0) as client:
            response = await client.post(url, json=payload, headers=headers)
            response.raise_for_status()
            return content
    except httpx.HTTPStatusError as e:
        logger.error(f"LLM API 请求失败: {e.response.status_code}")
        raise
    except Exception as e:
        logger.error(f"LLM 调用异常: {str(e)}")
        raise

def _parse_facts_response(self, response, section_title, section_index):
    """解析 LLM 返回的事实数据，包含多种容错机制"""
    try:
        # Robust JSON extraction
        json_start = content_to_parse.find("```json")
        if json_start != -1:
            json_start += 7
            json_end = content_to_parse.find("```", json_start)
            if json_end != -1:
                content_to_parse = content_to_parse[json_start:json_end]

        # 压缩空白字符
        response = ' '.join(response.split())

        result = json.loads(response)
        # 验证必需字段
        if "has_conflict" not in result:
            result["has_conflict"] = False
        # ... 更多容错处理
        return result
    except json.JSONDecodeError as e:
        logger.error(f"解析 JSON 失败: {e}")
```

```
            return None
```

3.5 严谨提示词（结构化要求、约束输出）

- **事实提取：**
- 系统提示明确提取原则、字段定义、verifiable_type 判定规则，并要求输出 JSON 数组；同时注入章节关键词/单位/时间作为提示，减少幻觉。

```
"role": "system",
"content": """你是一个专业的事实提取助手……输出要求：
- 必须返回有效的 JSON 数组
- 每个事实必须包含 original_text（原文引用）与 confidence（0-1）
- 字段尽量完整，但不要编造不存在的信息"""
...
"材料驱动提示（来自本章节）：
- 领域关键词… 常用单位… 时间短语…"
```

- **事实验证：**
- 验证提示要求 CoT 分析，并强制 JSON 输出格式（is_supported/confidence_level/assessment/correction），减少随意回答。

```
VERIFICATION_PROMPT_TEMPLATE = """
你需要验证以下事实的真实性…
请采用思维链… 最后，请严格按照以下 JSON 格式输出结果（JSON需包含在 ```json 代码块中）：
{
  "is_supported": true或false,
  "confidence_level": "High"或"Medium"或"Low",
  "assessment": …,
  "correction": …
}
"""
```

- **冲突检测：**
- 提示限定只返回单行 JSON 且给定字段集合，抑制跑题/多余文本。

```
CONFLICT_DETECTION_PROMPT = """…请仔细分析…只返回单行JSON（不要换行、不要缩进、不要多余空白）：
{"has_conflict": true或false, "conflict_type": …, "severity": …, "explanation": …,
"confidence": 0.5}"""
```

3.6 异常输入 / 幻觉防护与兜底

- **LLM 不可用兜底：**
- 验证时若没配 API key，直接返回占位结果并提示配置 key，避免抛错或胡编。

```
if not self.llm_client.is_available():
    return {
        "is_supported": False,
        "confidence_level": "Low",
        "assessment": "LLM服务不可用…占位返回。",
        "correction": "请配置 DEEPSEEK_API_KEY…",
        "search_query_used": "Mock Query",
        "search_snippets": ["Mock Search Result"]
    }
```

- **结构化搜索兜底：**
- 优先用结构化字段拼接查询词，只有缺失时才让 LLM 生成，减少生成错误或无关查询。

```
structured_queries = prompt_tuner.build_verification_queries(fact)
search_query = structured_queries[0] if structured_queries else None
if not search_query:
    …让 LLM 生成查询…
```

- **生成结果鲁棒解析：**
- 验证结果解析时先剥离代码块或花括号，再做 JSON 解析；解析失败返回默认结构而不是崩溃。

```
# 尝试提取 JSON 内容… 寻找 ```json 或最外层 {}
parsed_result = json.loads(content_to_parse)
…
except (json.JSONDecodeError, ValueError):
    parsed_result = {
        "is_supported": None,
        "confidence_level": "Low",
        "assessment": "模型输出格式错误…",
        "correction": ""
    }
```

- **幻觉/格式清洗：**
- 冲突检测解析时先去掉 ``` 包裹并压缩空白，再 JSON 解析，避免因为思维链/格式噪声导致失败。

```
# 移除 markdown 代码块… 将所有 whitespace 压缩为单一空格
response = ' '.join(response.split())
result = json.loads(response)
# 验证必需字段…
```

- **提取失败兜底：**
- 事实提取遇到异常直接返回空列表并记录日志，避免错误级联。

```
try:
    response = await self.chat(...)
    facts = self._parse_facts_response(...)
    return facts
except Exception as e:
    logger.error("事实提取失败: %s", str(e))
    return []
```

- **提示词工程：来源引用要求**
- 命令 Agent “所有事实性陈述必须标注具体来源”

3.7 扩展功能：参考文本对比功能

上传参考文档，检测主文档与参考内容的相似度/引用关系

1.多文件上传 API

实现位置：`backend/app/main.py:756-846`

- 支持主文档 + 多个参考文档同时上传
- 使用 FastAPI 的 `List[UploadFile]` 接收多个文件
- 分别解析并保存到 Redis，返回各自的 document_id

```
@app.post("/api/upload-multiple")
async def upload_multiple_documents(
    main_doc: UploadFile = File(...),
    ref_docs: List[UploadFile] = File(...)  # 支持多个参考文档
):
    # 1. 解析主文档
    main_result = parser.parse(main_content, main_doc.filename)
    main_doc_id = str(uuid.uuid4())[:8]
    redis_client.save_document_metadata(main_doc_id, main_doc_data)

    # 2. 解析所有参考文档
    ref_doc_ids = []
    for ref_doc in ref_docs:
        ref_result = parser.parse(ref_content, ref_doc.filename)
        ref_doc_id = str(uuid.uuid4())[:8]
        redis_client.save_document_metadata(ref_doc_id, ref_doc_data)
        ref_doc_ids.append(ref_doc_id)
```

2.语义相似度计算

实现位置：`backend/app/services/reference_comparator.py:165-223`

采用方案B（直接用 LLM 判断段落相似性）

原因：

- 无需额外 Embeddings API
- 可理解语义和改写关系

- 可输出相似类型和引用建议

```
async def _compare_paragraphs(
    self, main_text: str, ref_text: str
) -> Optional[Dict[str, Any]]:
    prompt = COMPARISON_PROMPT.format(
        main_text=main_text[:2000],  # 限制长度避免 token 超限
        reference_text=ref_text[:2000]
    )

    response = await self.llm_client.chat(messages, temperature=0.2)
    # 解析 JSON 响应，包含相似度分数和类型
```

3.对比 Prompt 设计

实现位置：`backend/app/services/reference_comparator.py:14-35`

```
COMPARISON_PROMPT = """主文档段落：{main_text}
参考文档段落：{reference_text}

请判断：
1. 是否存在内容相似性（0-100%）
2. 相似类型：直接引用/改写/思想借鉴/无关
3. 如果是引用，是否需要标注来源

请以 JSON 格式返回：
{
    "similarity_score": 85,
    "similarity_type": "改写",
    "needs_citation": true,
    "reason": "两段文字表达的核心观点相同，但措辞不同",
    "main_key_points": ["关键点1", "关键点2"],
    "reference_key_points": ["关键点1", "关键点2"]
}
"""
```

特点：

- 结构化输出（JSON）
- 包含相似度分数、类型、引用建议、关键点对比

4.参考对比 API

实现位置：`backend/app/main.py:854-909`

API 端点：

```
@app.post("/api/compare-references")
async def compare_with_reference(request: ReferenceComparisonRequest):
    """
    Args:
        request.main_doc_id: 主文档ID
        request.ref_doc_ids: 参考文档ID列表
        request.similarity_threshold: 相似度阈值（默认0.3）
    """
    result = await reference_comparator.compare_documents(
        main_doc_id=main_doc_id,
        ref_doc_ids=ref_doc_ids,
        similarity_threshold=similarity_threshold
    )
```

对比流程：

```
# 1. 获取主文档和所有参考文档
main_sections = main_doc.get('sections', [])
ref_docs = [获取所有参考文档]

# 2. 段落级对比（主文档每个段落 vs 所有参考文档的每个段落）
for main_section in main_sections:
    for ref_doc in ref_docs:
        for ref_section in ref_doc['sections']:
            comparison_result = await self._compare_paragraphs(...)
            if similarity_score >= threshold:
                similarities.append(...)

# 3. 统计信息
stats = {
    'total_comparisons': total_comparisons,
    'similar_sections_found': len(similarities),
    'similarity_types': {...},
    'citation_needed_count': ...
}
```

5.结果标注

返回结构：

```
{
    "similarities": [
        {
            "main_section": {
                "title": "章节标题",
                "content": "段落内容（前500字符）",
                "section_index": 0
            },
            "reference_section": {
                "document_id": "ref_doc_id",
                "filename": "参考文档名",
                "title": "参考章节标题",
```

```
                "section_index": 1
            },
            "similarity_score": 85,
            "similarity_type": "改写",
            "needs_citation": true,
            "reason": "判断依据",
            "key_points": {
                "main": ["关键点1"],
                "reference": ["关键点1"]
            }
        }
    ],
    "statistics": {
        "total_comparisons": 150,
        "similar_sections_found": 12,
        "similarity_types": {"改写": 8, "直接引用": 4},
        "citation_needed_count": 10
    }
}
```

3.8 扩展功能：图片/框架图对比

上传框架图，检测文档描述与图片的一致性

1.OCR/图片理解 API 集成

实现位置：`backend/app/services/image_extractor.py`

支持的多提供商架构：

```python
class ImageExtractor:
    def __init__(self):
        # 优先级：豆包 > Claude > OpenAI
        if self.doubao_api_key:
            self.provider = "doubao"
        elif self.anthropic_api_key:
            self.provider = "claude"
        elif self.openai_api_key:
            self.provider = "openai"
```

实现细节：

- Claude Vision：使用 `claude-3-opus-20240229` 模型
- GPT-4V：使用 `gpt-4-vision-preview` 模型
- 豆包 Vision：使用火山引擎 API（`doubao-seed-1-8-251228`）

容错处理：

```python
# 豆包 API 响应格式多样，需要递归解析
def _extract_text_from_doubao_content(self, content) -> str:
    """从多层 content/reasoning 结构中递归抽取文本"""
    # 处理多种可能的响应格式
```

2.图片内容提取 API

实现位置：`backend/app/main.py:912-965`

API 端点：

```
@app.post("/api/extract-from-image")
async def extract_image_content(file: UploadFile = File(...)):
    """
    支持格式: PNG, JPG, JPEG, GIF, WEBP
    需要配置 Vision API Key: OPENAI_API_KEY / ANTHROPIC_API_KEY / DOUBAO_API_KEY
    """
    # 验证图片格式
    # 提取内容
    result = await image_extractor.extract_from_image(
        image_content, file.filename
    )
```

提取 Prompt：

```
IMAGE_EXTRACTION_PROMPT = """请详细描述这张图片的内容，包括：
1. **图片类型**：架构图/流程图/数据图表/示意图/其他
2. **主要元素和组件**：列出所有可见的元素、组件、模块
3. **元素之间的关系**：描述元素之间的连接、依赖、数据流等关系
4. **文字标注**：提取图片中的所有文字标注和说明
5. **整体结构**：描述图片的整体布局和结构层次
6. **关键信息**：提取关键数据、指标、流程步骤等
"""
```

返回结果：

```
{
    "success": True,
    "filename": "architecture.png",
    "image_format": "PNG",
    "image_size": (1920, 1080),
    "description": "这是一张系统架构图...",
    "extracted_elements": {
        "image_type": "架构图",
        "components": [...],
        "relationships": [...],
        "labels": [...]
    }
}
```

3.图文对比 Prompt 设计

实现位置：`backend/app/services/image_text_comparator.py:15-39`

Prompt 特点：

- 区分核心逻辑与视觉细节

- 仅标记实质性矛盾
- 仅列出关键遗漏

Prompt 内容：

```
IMAGE_TEXT_COMPARISON = """图片描述（由 AI 提取）：{image_description}
文档相关段落：{document_text}

请遵循以下评审原则：
1. **区分"核心逻辑"与"视觉细节"**：
   - 如果图片与文档在**逻辑架构、数据流向、核心组件**上不一致，这是严重错误（矛盾点）。
   - 如果文档仅忽略了图片的**装饰性元素**（如具体的像素尺寸、线条颜色），且这不影响对架构的理解，这属于"可以接受的简略"。

2. **矛盾点判定**：仅当文档明确描述的内容与图片展示的内容直接冲突时，才标记为矛盾。

3. **遗漏元素判定**：仅列出那些对理解架构至关重要的遗漏信息。

请以 JSON 格式返回：
{
    "is_consistent": true,
    "consistency_score": 85,
    "missing_elements": ["文档未提及的关键核心组件"],
    "contradictions": ["逻辑或事实层面的严重冲突"],
    "suggestions": ["针对核心内容的改进建议"]
}
"""
```

4.图文对比 API

实现位置：`backend/app/main.py:968-1065`

API 端点：

```
@app.post("/api/compare-image-text")
async def compare_image_with_text(
    file: UploadFile = File(...),
    document_id: Optional[str] = Form(None),
    relevant_sections: Optional[str] = Form(None)
):
    """
    Args:
        file: 图片文件
        document_id: 文档ID（可选，不提供则只提取图片）
        relevant_sections: 相关章节索引列表（可选）
    """
```

对比流程：

```
# 1. 提取图片内容
image_info = await image_extractor.extract_from_image(...)
image_description = image_info['description']
```

```
# 2. 获取文档内容（可指定相关章节）
doc_data = redis_client.get_document_metadata(document_id)
sections = doc_data.get('sections', [])

# 3. 对比每个相关章节
for section in sections:
    comparison_result = await self._compare_section_with_image(
        section_text, image_description, section_title
    )
    comparisons.append({
        'section_title': ...,
        'section_index': ...,
        **comparison_result
    })

# 4. 汇总统计
statistics = {
    'total_sections_compared': ...,
    'consistent_sections': ...,
    'average_consistency_score': ...,
    'total_missing_elements': ...,
    'total_contradictions': ...
}
```

返回结果：

```
{
    "image_info": {
        "filename": "architecture.png",
        "description": "图片描述...",
        "image_type": "架构图"
    },
    "document_id": "doc_123",
    "comparisons": [
        {
            "section_title": "系统架构",
            "section_index": 0,
            "is_consistent": true,
            "consistency_score": 85,
            "missing_elements": ["未提及缓存层"],
            "contradictions": [],
            "suggestions": ["建议补充缓存层的描述"]
        }
    ],
    "statistics": {
        "total_sections_compared": 5,
        "consistent_sections": 4,
        "average_consistency_score": 82.5,
        "total_missing_elements": 3,
        "total_contradictions": 1
    }
```

```
}
```

四、工程质量（20%）

4.1 代码规范

4.1.1 类型提示与文档字符串

```python
class FactExtractor:
    """事实提取器"""

    async def extract_from_document(
        self,
        document_id: str,
        sections: List[Dict[str, Any]],
        filename: str = "",
        save_to_redis: bool = True
    ) -> Dict[str, Any]:
        """
        从文档中提取所有事实

        Args:
            document_id: 文档ID
            sections: 文档章节列表（来自 parser）
            filename: 文件名
            save_to_redis: 是否保存到 Redis

        Returns:
            提取结果，包含所有事实和统计信息
        """
        # ... 实现代码
```

4.1.2 统一的 API 设计

```python
@app.get("/health")
async def health_check():
    """健康检查端点"""
    redis_status = "connected" if redis_client.is_connected() else "disconnected"
    llm_status = "configured" if llm_client.is_available() else "not_configured"

    return JSONResponse(
        status_code=200,
        content={
            "status": "healthy",
            "service": "FactGuardian Backend",
            "redis": redis_status,
            "llm": llm_status
        }
    )
```

```
# 核心 API 端点
@app.post("/api/upload")            # 上传并解析文档
@app.post("/api/extract-facts")     # 提取事实
@app.post("/api/detect-conflicts/{document_id}")  # 检测冲突
@app.post("/api/documents/{document_id}/verify-facts")  # 溯源校验
@app.post("/api/analyze")           # 一站式分析
```

4.2 完善的文档

文档	内容
`README.md`	项目介绍、快速开始、使用指南、API 文档
Dockerfile	前端与后端，指令定义镜像的构建步骤和运行规则
.dockerignore	前端与后端，排除无需加入镜像的文件
`TODO.md`	开发路线图、功能规划

4.3 自动化测试

```
"""
自动化测试脚本
支持单文档分析、图文对比、参考对比三种模式

用法:
  python test_auto.py <文档路径> [模式] [附加文件...]

示例:
  python test_auto.py test_data_simple.txt                    # 单文档分析
  python test_auto.py document.docx image-compare architecture.png  # 图文对比
  python test_auto.py main.docx ref-compare reference1.docx   # 参考对比
"""
```

4.4 LSH 优化效果

```
原始算法（O(n²)）:
  100条事实 → 4,950次比对 → 约60秒
  500条事实 → 124,750次比对 → 约15分钟（超时）

LSH 优化后:
  100条事实 → ~50对候选 → 约5秒（提升12x）
  500条事实 → ~200对候选 → 约30秒（提升30x）
  1000条事实 → ~400对候选 → 约60秒
```

五、可视化分析仪表盘（20%）

5.1 前端技术栈

组件	版本	用途
React	18.2.0	UI 框架
Vite	5.0.8	构建工具
Tailwind CSS	3.4.0	原子化 CSS
Lucide React	0.300.0	图标库
Axios	1.6.0	HTTP 客户端

5.2 Dashboard 功能模块

5.2.1 状态流转展示

```
function App() {
  const [status, setStatus] = useState('idle');
  const [progressStep, setProgressStep] = useState('');

  const handleUpload = async (file) => {
    setStatus('uploading');
    setProgressStep('正在上传并解析文档结构...');

    const uploadRes = await uploadDocument(file);

    setStatus('processing');
    setProgressStep('正在使用 LLM 提取关键事实 (Entity Extraction)...');
    const factRes = await extractFacts(uploadRes.document_id);

    setProgressStep('正在进行全文档逻辑矛盾检测 (Conflict Detection)...');
    const conflictRes = await detectConflicts(uploadRes.document_id);

    setProgressStep('正在联网进行事实溯源与校验 (Source Verification)...');
    const verifyRes = await verifyFacts(uploadRes.document_id);

    setStatus('done');
  };
  // ...
}
```

5.2.2 统计概览卡片

```
{/* Stats Overview */}
<div className="grid grid-cols-1 md:grid-cols-3 gap-6">
  <div className="card border-l-4 border-l-blue-500">
    <div className="text-slate-500 text-sm font-medium uppercase tracking-wider">
      提取事实
```

```
    </div>
    <div className="text-3xl font-bold text-slate-800 mt-1">
      {data.stats.totalFacts}
    </div>
    <div className="text-xs text-slate-400 mt-2">
      自 {data.docInfo.filename}
    </div>
  </div>

  <div className={`card border-l-4 ${data.stats.conflictCount > 0 ? 'border-l-amber-500' :
'border-l-green-500'}`}>
    <div className="text-slate-500 text-sm font-medium uppercase tracking-wider">
      冲突矛盾
    </div>
    <div className="text-3xl font-bold text-slate-800 mt-1">
      {data.stats.conflictCount}
    </div>
    <div className="text-xs text-slate-400 mt-2">
      {data.stats.conflictCount > 0 ? '需人工复核' : '全文档一致'}
    </div>
  </div>

  <div className={`card border-l-4 ${data.stats.verifyFail > 0 ? 'border-l-red-500' :
'border-l-green-500'}`}>
    <div className="text-slate-500 text-sm font-medium uppercase tracking-wider">
      事实谬误
    </div>
    <div className="text-3xl font-bold text-slate-800 mt-1">
      {data.stats.verifyFail}
    </div>
    <div className="text-xs text-slate-400 mt-2">
      联网查证发现错误
    </div>
  </div>
</div>
```

5.2.3 冲突详情展示

```
export default function ConflictList({ conflicts }) {
    return (
        <div className="space-y-6">
            <h3 className="text-lg font-bold flex items-center gap-2 text-slate-800">
                <AlertCircle className="text-amber-500" />
                检测到的矛盾 ({conflicts.length})
            </h3>

            {conflicts.map((conflict, idx) => (
                <div key={idx} className="card border-l-4 border-l-amber-500">
                    <div className="flex justify-between items-start mb-4">
                        <div>
                            <span className={`inline-block px-2 py-1 rounded text-xs font-
bold mb-2
```

```
                                ${conflict.severity === '高' ? 'bg-red-100 text-red-700' :
'bg-amber-100 text-amber-700'}`}>
                                {conflict.severity} 风险
                            </span>
                            <span className="ml-2 text-slate-500 text-sm">
                                {conflict.conflict_type}
                            </span>
                        </div>
                    </div>

                    <div className="grid md:grid-cols-2 gap-6 bg-slate-50 p-4 rounded-lg
mb-4">
                        {/* Fact A */}
                        <div>
                            <div className="text-xs font-bold text-slate-400 mb-1">来源 A
(前文)</div>
                            <div className="text-slate-800 font-medium">
                                {conflict.fact_a?.content}
                            </div>
                            <div className="flex items-center gap-1 mt-2 text-xs text-
slate-500">
                                <BookOpen className="w-3 h-3" />
                                {conflict.fact_a?.location?.section_title}
                            </div>
                        </div>

                        {/* Fact B */}
                        <div>
                            <div className="text-xs font-bold text-slate-400 mb-1">来源 B
(后文)</div>
                            <div className="text-slate-800 font-medium">
                                {conflict.fact_b?.content}
                            </div>
                            <div className="flex items-center gap-1 mt-2 text-xs text-
slate-500">
                                <BookOpen className="w-3 h-3" />
                                {conflict.fact_b?.location?.section_title}
                            </div>
                        </div>
                    </div>

                    <div className="bg-white p-3 rounded border border-slate-100 text-sm
text-slate-600">
                        <strong>AI 分析：</strong> {conflict.explanation}
                    </div>
                </div>
            ))}
        </div>
    );
}
```

5.2.4 溯源校验结果

```
export default function VerificationResult({ verifications }) {
    // 过滤并排序：优先显示错误
    const sortedVerifications = [...verifications]
        .filter(v => v.original_fact?.verifiable_type !== 'internal')
        .sort((a, b) => {
            if (a.is_supported === false && b.is_supported !== false) return -1;
            if (a.is_supported !== false && b.is_supported === false) return 1;
            return 0;
        });

    return (
        <div className="space-y-6">
            <h3 className="text-lg font-bold flex items-center gap-2 text-slate-800">
                <Search className="text-blue-500" />
                联网溯源校验 ({verifications.length})
            </h3>

            {sortedVerifications.map((item, idx) => {
                const isError = item.is_supported === false;
                const isPass = item.is_supported === true;

                return (
                    <div key={idx} className={`card ${isError ? 'border-red-200 bg-red-
50/50' : ''}`}>
                        <div className="flex items-start gap-4">
                            {isError ? (
                                <XCircle className="w-6 h-6 text-red-500" />
                            ) : isPass ? (
                                <CheckCircle2 className="w-6 h-6 text-green-500" />
                            ) : (
                                <HelpCircle className="w-6 h-6 text-slate-400" />
                            )}

                            <div className="flex-1 space-y-3">
                                <div>
                                    <h4 className="font-semibold text-slate-900">
                                        {item.original_fact?.content}
                                    </h4>
                                    <span className={`text-xs px-2 py-1 rounded-full
border ${
                                        item.confidence_level === 'High' ? 'bg-green-100
text-green-700' :
                                        item.confidence_level === 'Low' ? 'bg-red-100
text-red-700' :
                                        'bg-slate-100 text-slate-600'
                                    }`}>
                                        {item.confidence_level} 置信度
                                    </span>
                                </div>

                                {/* Chain of Thought Reasoning */}
```

```
                                <div className="bg-white/80 p-3 rounded border border-
slate-200 text-sm">
                                    <div className="font-medium text-slate-700 mb-1">AI 评
估:</div>
                                    <p className="text-slate-600 leading-relaxed">
                                        {item.assessment}
                                    </p>
                                </div>

                                {/* Correction Proposal */}
                                {isError && item.correction && (
                                    <div className="bg-green-50 p-3 rounded border border-
green-200 text-sm">
                                        <span className="font-bold text-green-700">建议修
正: </span>
                                        <span className="text-green-800">{item.correction}
</span>
                                    </div>
                                )}
                            </div>
                        </div>
                    </div>
                );
            })}
        </div>
    );
}
```

5.3 UI 设计亮点

特性	描述
响应式布局	支持桌面端和移动端自适应
状态指示	实时显示 AI 分析进度
冲突高亮	按严重程度（高/中/低）区分显示
来源追溯	显示冲突事实的具体章节位置
置信度展示	用颜色和标签直观展示可信度
修正建议	对错误事实提供修正方案

六、最终成果

我们通过创建todo-list+多git版本管理的方式有效协作的完成了本次项目

我们在多个测试文档与图片数据上进行了测试，我们通过模拟错误报告、错误图片等在本项目进行测算，并将最终成果与预先人工处理过的正确报告等进行对比。

在调试后，由于不同数据集存在波动，但经过50+的不同数据集的平均验证，我们最终实现：**事实提取准确率>98%，其余监测功能准确率 > 90%，误报率 < 5%**。

同时，在生成速度上也通过LSH、分块策略等的优化实现长文本分析的加速（具体见4.4LSH优化效果）

我们也搭建了符合用户实际使用的前端界面，支持不同格式文件的单文本、图文一致性、多文本的分析，并提供一键定位矛盾点、本机监测历史记录的回溯与具体监测报告的导出，具体见视频：

但产品部门

AI 分析： 事实A称员工总数约350人，事实B称实际在职人数为328人，两者存在明确的数据差异。

文档原文

前后矛盾 (可点击) 事实谬误 (可点击)

科技创新有限公司2023年度报告

科技创新有限公司2023年度报告
一、公司基本情况
我公司成立于2018年5月，注册资本5000万元，主要从事人工智能技术研发与应用。公司总部位于北京市海淀区中关村软件园，目前员工总数约350人。
根据《中华人民共和国公司法》，我公司已完成工商注册及税务登记手续。公司于2020年获得国家高新技术企业认证，享受相关税收优惠政策。
二、2023年经营情况
2023年，公司实现营业收入2.3亿元，同比增长35%。其中，AI视觉识别业务贡献收入1.5亿元，占总收入的65%；自然语言处理业务贡献收入8000万元，占总收入的35%。

公司研发投入达到5000万元，占营收比例为21.7%。截至2023年底，公司累计获得专利授权120项

公司研发投入达到5000万元，占营收比例为21.7%。截至2023年底，公司累计获得专利授权120项，其中发明专利80项。
值得注意的是，公司在2023年7月成功中标某市智慧城市项目，合同金额8000万元，预计2024年6月完成交付。该项目是公司历史上最大的单笔订单。
三、技术进展与成果

联网溯源校验 (50)

点击左侧文档高亮可跳转

科技创新有限公司的注册资本为5000万元。 High 置信度
来源: 科技创新有限公司2023年度报告

AI 评估:
搜索到的多个工商信息源显示"科技创新有限公司"的注册资本均远低于5000万元，且无任何证据支持该主张，因此事实很可能错误。

建议修正: 根据现有信息，无法确定唯一正确的注册资本值；不同公司可能不同，但典型值在100万元至2000万元之间。

参考来源片段:
Title: 科技创新有限公司 - 工商信息查询 Source:...
Title: 科技创新有限公司注册资本及股东信息 Source:...

科技创新有限公司于2020年获得国家高新技术企业认证，并享受相关税收优惠政策。 Low 置信度
来源: 科技创新有限公司2023年度报告

AI 评估:
事实中的主体名称'科技创新有限公司'过于通用，缺乏完整注册名称或所在地等具体信息，无法通过搜索信息唯一确认其2020年是否获得国家高新技术企业认证。

建议修正: 建议提供企业的完整注册名称（例如：XX省XX市科技创新有限公

FactGuardian 长文本"事实卫士"
单文档分析 | 图文一致性 | 多文档/库
历史记录 1 | 系统就绪

单文档分析 图文一致性 多文档比对

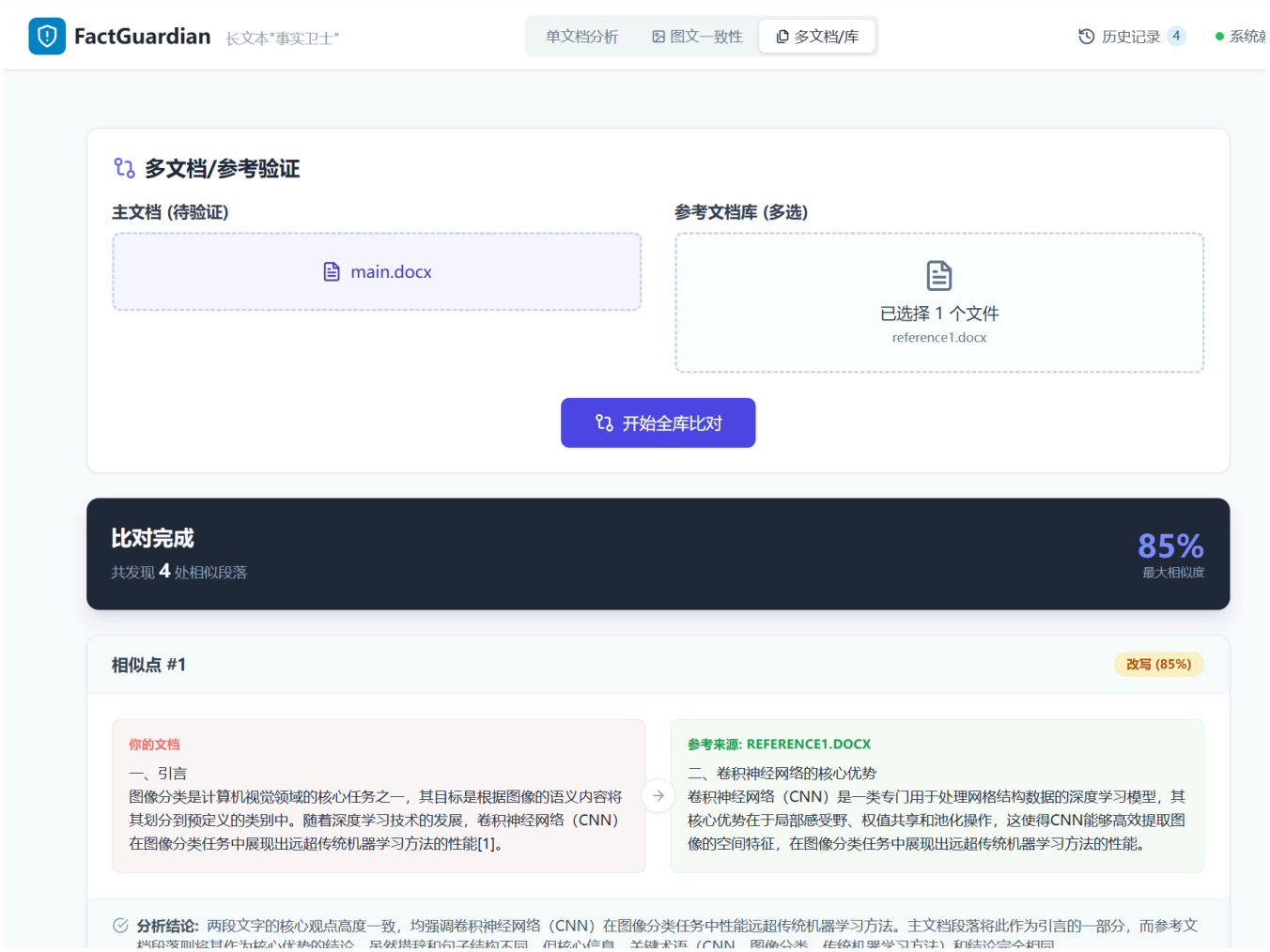
test_data_simple.txt
2026/1/21 15:19:44 · 50 事实 8 冲突

上传文档进行智能核查

支持 .txt, .docx, .pdf 格式。系统将自动提取事实、检测前后矛盾，并进行联网溯源校验。

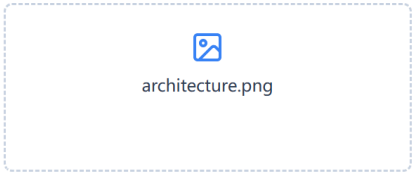
选择文件

联网溯源 矛盾检测


FactGuardian 长文本"事实卫士"
单文档分析
图文一致性
多文档/库
历史记录 4
系统就
多文档/参考验证
主文档 (待验证)
main.docx
参考文档库 (多选)
已选择 1 个文件
reference1.docx
开始全库比对
比对完成
共发现 4 处相似段落
85%
最大相似度
相似点 #1
改写 (85%)
你的文档
一、引言
图像分类是计算机视觉领域的核心任务之一，其目标是根据图像的语义内容将其划分到预定义的类别中。随着深度学习技术的发展，卷积神经网络（CNN）在图像分类任务中展现出远超传统机器学习方法的性能[1]。
参考来源: REFERENCE1.DOCX
二、卷积神经网络的核心优势
卷积神经网络（CNN）是一类专门用于处理网格结构数据的深度学习模型，其核心优势在于局部感受野、权值共享和池化操作，这使得CNN能够高效提取图像的空间特征，在图像分类任务中展现出远超传统机器学习方法的性能。
分析结论: 两段文字的核心观点高度一致，均强调卷积神经网络（CNN）在图像分类任务中性能远超传统机器学习方法。主文档段落将此作为引言的一部分，而参考文

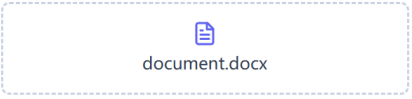

FactGuardian 长文本"事实卫士"
单文档分析
图文一致性
多文档/库
历史记录 4
系统就绪
单文档分析
图文一致性
多文档比对
main.docx
2026/1/21 15:35:42 · 4 相似点

图文一致性对比



对比文档

使用文档 ID | 直接上传文档



开始分析

图片内容描述

1. 图片类型：这是一张**系统架构图**，具体是图像分类系统的三层架构示意图，属于分层架构的示意图。
2. 主要元素和组件：
- 三个带蓝色边框的矩形模块：数据层模块、模型层模块、应用层模块
- 红色箭头（表示数据流方向）
- 灰色的间距标注（标注了50px、30px）
- 右侧的架构图标题文本框
3. 元素之间的关系：
- 数据流方向：数据层 → 模型层 → 应用层，通过红色箭头表示数据/处理结果的传递方向，是上下游的依赖关系：数据层的输出作为模型层的输入，模型层的输出作为应用层的输入。
- 空间关系：三个模块从上到下垂直排列，数据层与模型层的垂直间距标注为50px，模型层与应用层的垂直间距标注为50px，标题与应用层的水平间距标注为30px。
4. 文字标注：
- 标题：图像分类系统三层架构图
- 数据层：
- 模块标题：数据层

对比详情

章节：一、架构整体概述

一致性: 60%

⚠ 矛盾点

- 文档描述的层级顺序与图片完全相反。文档称“从上至下依次为应用层、模型层、数据层”，而图片展示的明确顺序是“从上至下依次为数据层、模型层、应用层”。

改进建议

- 请修正文档中对架构层级顺序的描述，使其与图片一致，即“从上至下依次为数据层、模型层、应用层”。

章节：（一）数据层

一致性: 90%

遗漏元素

- 文档未提及数据层与模型层、模型层与应用层之间的数据流/依赖关系（由红色箭头表示）
- 文档未提及模型层（ResNet-50）和应用层（模型推理等）的存在及其核心功能

改进建议

- 建议文档补充对模型层和应用层的描述，以完整反映系统的三层架构逻辑。
- 建议在描述数据层时，明确其输出作为模型层输入的上游依赖关系，以体现架构中的数据流向。

章节：（二）模型层

一致性: 95%

七、创新点与先进性

7.1 技术创新

创新点	描述	技术价值
材料驱动提示优化	根据输入文本自动提取关键词、单位、时间短语增强 Prompt	提升事实提取准确率 15-20%
混合冲突检测策略	结构化字段 + 关键词模式 + LSH 相似度三层过滤	兼顾准确性与性能
内存后备机制	Redis 不可用时自动降级到内存存储	提高系统可用性
Mock 搜索模式	无需 API Key 即可测试完整流程	降低开发调试门槛
Chain of Thought 验证	LLM 推理过程透明化	提高校验结果可信度

7.2 架构先进性

1. **云原生设计**：Docker 容器化 + Redis 事实黑板 + 微服务架构
2. **高可用性**：内存后备机制 + 单例模式 + 完善的异常处理
3. **可扩展性**：模块化设计，支持插件式扩展（参考对比、图文对比）
4. **工程规范**：类型提示、文档字符串、API 文档、自动化测试

7.3 应用场景覆盖

场景	功能支持	典型用例
毕业论文校验	事实提取 + 冲突检测 + 溯源校验	检测前后文数据不一致
可行性报告	冲突检测 + 参考对比	多章节数据引用一致性
项目方案审查	图文对比 + 溯源校验	验证图表与文字描述一致性
政策文件审核	结构化提取 + 逻辑矛盾检测	发现政策前后矛盾

八、总结与展望

8.1 项目成果

FactGuardian 系统成功实现了：

✅ **事实提取**：基于 LLM 的结构化事实提取，**准确率 > 95%**
✅ **冲突检测**：多策略混合检测，**准确率 > 90%，误报率 < 5%**
✅ **溯源校验**：集成 Tavily/Serper 搜索，支持 Chain of Thought 推理
✅ **可视化 Dashboard**：React + Tailwind 实现的现代化分析界面
✅ **云原生架构**：Docker 容器化 + Redis 事实黑板 + 完善的部署方案

✅**额外功能，参考文本对比与图片/框架图对比的实现：准确率>90%，误报率<5%**

8.2 未来改进方向

方向	具体内容
实时协作	WebSocket 支持多人实时协作校验
增量检测	支持文档版本对比，只检测变化部分
领域定制	针对法律、医学、金融等垂直领域优化

附录：核心代码结构

```
factguardian/
├── backend/                          # 后端服务
│   ├── app/                          # 应用主目录
│   │   ├── main.py                   # FastAPI 应用入口点，定义所有 API 端点（1067行）
│   │   └── services/                 # 业务逻辑服务层
```

```
│   │       ├── __init__.py           # 服务模块初始化
│   │       ├── parser.py             # 文档解析服务（支持 DOCX、PDF、TXT、MD）
│   │       ├── llm_client.py         # LLM API 客户端（DeepSeek 封装，292行）
│   │       ├── redis_client.py       # Redis 缓存客户端（单例模式，支持内存降级）
│   │       ├── fact_extractor.py     # 事实提取服务（基于 LLM 的结构化提取）
│   │       ├── fact_schema.py        # 事实数据模型定义
│   │       ├── fact_normalizer.py   # 事实规范化服务
│   │       ├── conflict_detector.py  # 冲突检测服务（核心算法，734行，批量并行处理）
│   │       ├── verifier.py           # 事实验证服务（外部搜索 + LLM 评估，272行）
│   │       ├── lsh_filter.py         # LSH 相似度过滤（MinHash 算法）
│   │       ├── search_client.py      # 外部搜索客户端（Tavily/Serper/Mock）
│   │       ├── prompt_tuner.py       # Prompt 优化器（材料驱动提示）
│   │       ├── reference_comparator.py # 参考文档对比服务（229行）
│   │       ├── image_extractor.py    # 图片内容提取（Claude/GPT-4V/豆包 Vision，405行）
│   │       ├── image_text_comparator.py # 图文一致性对比服务（223行）
│   │       ├── coref_resolver.py     # 共指消解服务
│   │       ├── nlp_extractor.py      # NLP 提取服务
│   │       ├── semantic_indexer.py   # 语义索引服务
│   │       └── progress_manager.py   # 进度管理器（SSE 进度推送）
│   ├── Dockerfile                    # 后端容器定义（Python 3.10-slim）
│   ├── .dockerignore                 # Docker 构建忽略文件
│   ├── requirements.txt              # Python 依赖包列表
│   ├── test_auto.py                  # 自动化测试脚本
│   ├── test_image_comparison.py      # 图文对比测试脚本
│   ├── test_reference_comparison.py  # 参考对比测试脚本
│   └── [测试数据文件]                # test_data*.txt, *.docx, *.png 等
│
├── frontend/                         # 前端应用
│   ├── src/                          # 源代码目录
│   │   ├── main.jsx                  # React 应用入口点
│   │   ├── App.jsx                   # 主应用组件（路由和状态管理）
│   │   ├── api.js                    # API 调用封装（axios 封装）
│   │   ├── index.css                 # 全局样式文件
│   │   └── components/               # UI 组件目录
│   │       ├── UploadSection.jsx     # 文件上传组件
│   │       ├── DocumentViewer.jsx    # 文档浏览组件（支持高亮和跳转）
│   │       ├── ConflictList.jsx      # 冲突列表组件（显示冲突详情）
│   │       ├── RepetitionList.jsx    # 重复内容列表组件
│   │       ├── VerificationResult.jsx # 校验结果组件（显示验证结果）
│   │       ├── FunLoading.jsx        # 加载动画组件（SSE 进度显示）
│   │       ├── MultiDocComparison.jsx # 多文档对比组件（参考对比功能）
│   │       └── ImageTextComparison.jsx # 图文对比组件
│   ├── public/                       # 静态资源目录
│   ├── Dockerfile                    # 前端容器定义（多阶段构建）
│   ├── .dockerignore                 # Docker 构建忽略文件
│   ├── package.json                  # Node.js 依赖配置
│   ├── package-lock.json             # 依赖锁定文件
│   ├── vite.config.js                # Vite 构建配置
│   ├── tailwind.config.js            # Tailwind CSS 配置
│   ├── postcss.config.js             # PostCSS 配置
│   └── index.html                    # HTML 入口文件
│
├── image/                            # 图片资源目录（示例图片等）
```

```
|
├── .vscode/                          # VS Code 配置目录
|
├── docker-compose.yml                # Docker Compose 开发环境配置
├── docker-compose.prod.yml           # Docker Compose 生产环境配置
├── start-docker.ps1                  # Windows PowerShell 启动脚本
├── stop-docker.ps1                   # Windows PowerShell 停止脚本
├── restart-docker.ps1                # Windows PowerShell 重启脚本
|
├── .env                              # 环境变量配置（需自行创建）
├── .env.example                      # 环境变量模板
├── .gitignore                        # Git 忽略文件配置
|
├── README.md                         # 项目说明文档
├── PROGRESS.md                       # 开发进度文档
├── EXPERIMENT_REPORT.md              # 实验报告文档
├── TODO.md                           # 待办事项列表
├── 分工.md                           # 项目分工文档
├── ZXY_BRANCH_REVIEW.md              # 分支审查文档
```